FLORIDA POLYTECHNIC UNIVERSITY

COP 4020: Programming Languages

Describing Syntax: BNF Grammar

Syntax vs semantic

- Syntax: the form or structure of the expressions, statements, and program units
- Semantics: the meaning of the expressions, statements, and program units
- Syntax and semantics provide a language's definition

Example: Java while statement

- Syntax

  while (boolean_expr) statement

- Sematic
  - when the current value of the Boolean expression is true, the embedded statement is executed. Then control implicitly returns to the Boolean expression to repeat the process. If the Boolean expression is false, control transfers to the statement following the while construct.

Terminology

- A sentence is a string of characters over some alphabet
- A language is a set of sentences
- A lexeme is the lowest level syntactic unit of a language such as operators and special words (e.g., *, sum, begin)
- A token is a category of lexemes (e.g., identifier)

Example

- Consider the following Java statement:

  index = 2 * count + 17;

- The lexemes and tokens of this statement are

Lexemes	*Tokens*
`index`	identifier
`=`	equal_sign
`2`	int_literal
`*`	mult_op
`count`	identifier
`+`	plus_op
`17`	int_literal
`;`	semicolon

Language recognizers vs generators

- Recognizer
  - Focuses on analyzing existing code to determine its adherence to the syntax rules of a programming language

- Generator
  - Focuses on creating or generating new code based on higher-level specifications

Formal methods of describing syntax

- Grammars are used to describe the syntax of PLs

- Backus-Naur Form (BNF)
  - A notation that is widely used to express the syntax of PLs
  - It was introduced by John Backus and Peter Naur in the late 1950s and early 1960s as a way to describe the syntax of ALGOL 60.
  - BNF consists of a set of production rules that describe how valid strings in the language can be formed

BNF fundamentals

- Each rule has the following form
  LHS → RHS

- LHS (left-hand side) is an abstraction and is a nonterminal symbol
  - A nonterminal symbol is replaced by one or more sequences of other terminal and/or nonterminal symbols

- → means "is defined as" or "can be replaced by"

- RHS (right-hand side) represents a sequence of terminals (actual symbols of the language; the lexemes) and/or non-terminals

Examples of rules

- A simple Java assignment statement

<assign> → <var> = <expression>

- Nonterminal symbols can have two or more distinct definitions
  - Example: Java if statement can be described with the rules

<if_stmt> → **if** (<logic_expr>) <stmt>
<if_stmt> → **if** (<logic_expr>) <stmt> **else** <stmt>

or with the rule

<if_stmt> → **if** (<logic_expr>) <stmt>
| **if** (<logic_expr>) <stmt> **else** <stmt>

  - The symbol | means logical OR

Recursive rule

- A rule is recursive if its LHS appears in its RHS
- Example

<ident_list> → identifier
| identifier, <ident_list>

Grammers and derivations

- A grammar is a generative device for defining languages.
- The sentences of the language are generated through a sequence of applications of the rules, beginning with a special nonterminal of the grammar called the start symbol
- This sequence of rule applications is called a derivation

Example: A grammar for a small language

<program> → **begin** <stmt_list> **end**

<stmt_list> → <stmt>
| <stmt> ; <stmt_list>

<stmt> → <var> = <expression>

<var> → A | B | C

<expression> → <var> + <var>
| <var> – <var>
| <var>

An example of derivation of a program in this language

<program> → **begin** <stmt_list> **end**

<stmt_list> → <stmt>
| <stmt> ; <stmt_list>

<stmt> → <var> = <expression>

<var> → A | B | C

<expression> → <var> + <var>
| <var> – <var>
| <var>

<program> => **begin** <stmt_list> **end**
=> **begin** <stmt> ; <stmt_list> **end**
=> **begin** <var> = <expression> ; <stmt_list> **end**
=> **begin** A = <expression> ; <stmt_list> **end**
=> **begin** A = <var> + <var> ; <stmt_list> **end**
=> **begin** A = B + <var> ; <stmt_list> **end**
=> **begin** A = B + C ; <stmt_list> **end**
=> **begin** A = B + ; <stmt> **end**
=> **begin** A = B + C ; <var> = <expression> **end**
=> **begin** A = B + C ; B = <expression> **end**
=> **begin** A = B + C ; B = <var> **end**
=> **begin** A = B + C ; B = C **end**

Example: A grammar for simple assignment statements

<assign> → <id> = <expr>
<id> → A | B | C
<expr> → <id> + <expr>
| <id> * <expr>
| (<expr>)
| <id>

Derivation example

<assign> → <id> = <expr>
<id> → A | B | C
<expr> → <id> + <expr>
| <id> * <expr>
| (<expr>)
| <id>

- Leftmost derivation of statement

  A = B * (A + C)

<assign> => <id> = <expr>
=> A = <expr>
=> A = <id> * <expr>
=> A = B * <expr>
=> A = B * (<expr>)
=> A = B * (<id> + <expr>)
=> A = B * (A + <expr>)
=> A = B * (A + <id>)
=> A = B * (A + C)

Exercises

- Consider this BNF grammar:

<assign> → <id> = <expr>
<id> → A | B | C
<expr> → <id> + <expr>
| <id> * <expr>
| (<expr>)
| <id>

- Perform rightmost derivation of the statement, A = B * (A + C)

- Which of the following statements are in the language generated by this grammar?
  - A = B
  - A = D
  - A = (B)
  - A = B – C
  - A = (B) + A
  - A = (B + C) * (A + B)